Jaql
Running Pipes in the Clouds

Kevin Beyer, Vuk Ercegovac, Eugene Shekita, Jun Rao, Ning Li, Sandeep Tata
IBM Almaden Research Center
http://code.google.com/p/jaql/
Motivating Scenarios

- **Driving forces**
  - **Data characteristics**
    - Unstructured/Semi-structured (e.g., call-center records, pathology reports, ..)
    - Dynamic & Continuously Evolving (e.g., click streams, application logs, system logs)
    - Massive Scale (several terabytes to petabytes ..)
  - **Nature of analytics**
    - Compute intensive
    - Evolving
      - Demands a **flexible** platform to dynamically accommodate new analytic flows
ES2: Intranet Data → Semantic Search

Import

- Crawl

Analyze

- Information Extraction & Mining

Transform

- Generate indexing artifacts
- Indexing

Export

- ES2 Semantic Search

Cloud Infrastructure

Lucene Indexes

IBM Almaden Research Center
ES2: Intranet Data → Semantic Search
ES2’s Analysis Stage

Local Analysis:
- **System T** extracts structured information, e.g., candidate home pages with person name

Global Analysis:
- **Jaql** finds the best pages within groups of pages, e.g., personal homepage

Mining:
- Use **map-reduce** for mining
  - Acronym extraction
  - Geo-classification
  - Learning regular expressions
Goals for Jaql

- Semi-structured data manipulation
  - Use JSON as a data model
    - Data is converted to/from JSON view
    - Most data has a natural JSON representation
      - Relational tables, CSV, XML, …

- Easily extended
  - E.g., Java, Python, JavaScript, …

- Exploit massive parallelism using Hadoop
  - Queries compiled to map-reduce jobs
JSON Examples

[] == array, {} == record or object, xxx: == field name

[  
  { from: 17, to: 19, msg: "...",  
    phones: ["+1-415-555-1212", "+1-408-555-1234"], names:["Jill","Jenny"] },
  { from: 17, to: 12, msg: "...", names:["Jack","Jill"]},
  { from: 17, to: 12, msg: "...", phones: ["+1-415-555-2345"] }
]

[  
  { id: 12, name: "Joe Smith", bday: date("1971-03-07"), zip: 94114 },
  { id: 19, name: "Alicia Fox", bday: date("1975-04-20"), zip: 94114 }
]

[  
  { city: "San Francisco", zips: [94110,94112,94114], area codes: [415 ] },
  { city: "San Jose", zips: [95120,95123], area codes: [408] }
]
Writing a pipeline in Jaql

Find users in zip 94114

```json
[
  { id: 12, name: "Joe Smith", bday: date("1971-03-07"), zip: 94114 },
  { id: 19, name: "Alicia Fox", bday: date("1975-04-20"), zip: 94114 }
]
```

```json
[
  { id: 12, name: "Joe Smith" },
  { id: 19, name: "Alicia Fox" }
]
Writing a pipeline in Jaql: read

Find users in zip 94114

Query

```
read("hdfs:users");
```

Data

```
[
  { id: 12, name: “Joe Smith”, bday: date(“1971-03-07”), zip: 94114 },
  { id: 19, name: “Alicia Fox”, bday: date(“1975-04-20”), zip: 94114 }
]
```
Writing a pipeline in Jaql: filter

Find users in zip 94114

Query
read("hdfs:users")
-> filter $.zip == 94114;

Data
[
  { id: 12, name: "Joe Smith",
    bday: date("1971-03-07"), zip: 94114 },
  { id: 19, name: "Alicia Fox",
    bday: date("1975-04-20"), zip: 94114 }
]
Writing a pipeline in Jaql: transform

Find users in zip 94114

**Query**

```jaql
read("hdfs:users")
-> filter $.zip == 94114
-> transform { $.id, $.name };```

**Data**

```
[{
  id: 12, name: “Joe Smith”
},
{
  id: 19, name: “Alicia Fox”
}]
```
Writing a pipeline in Jaql: write

Find users in zip 94114

**Query**

```jaql
read("hdfs:users")
-> filter $.zip == 94114
-> transform { $.id, $.name }
-> write("hdfs:inzip");
```

**Data**

```
[
  { id: 12, name: "Joe Smith" },
  { id: 19, name: "Alicia Fox" }
]
```
**JAQL I/O**

- I/O layer abstracts details of data location + format
  - Examples of data stores:
    - HDFS, HBase, Amazon’s S3, local FS, HTTP request, JDBC call
  - Examples of data formats:
    - JSON text, CSV, XML (default is JSON binary)
- JSON used as a data model
Storing Files in HDFS

- **File is divided into blocks**
  - Default block is 64MB
- **Blocks are replicated on HDFS nodes**
  - Default replication is three times
- **The name node tracks file names and block locations**
Map Jobs in Hadoop

- **JobConf**
  - Specifies job configuration

- **InputFormat**
  - Abstract data source
  - Creates Splits and RecordReader

- **Split**
  - Abstract notion of a data chunk
  - Each split handled by a map task
  - Contains location hints

- **RecordReader**
  - Produces stream of values from a Split

- **Map Task**
  - Processes values in one Split
  - Calls user map code
  - Reevaluated on error

- **OutputFormat**
  - Abstract data sink

- **Infrastructure handles**
  - Parallelism
  - Load balancing
  - Fault-tolerance
Jaql using Map

read("hdfs:users")
  -> filter $\.zip == 94114
  -> transform \{ $\.id, $\.name \}
  -> write("hdfs:inzip");

Equivalent map-reduce job in Jaql

mapReduce(
  { input : { uri: "hdfs:users" },
    map : fn($) ( $ -> filter $\.zip == 94114
                   -> transform \{ $\.id, $\.name \} ),
    output : { uri: "hdfs:inzip" } })
User-defined Functions

Identify phone numbers in each message

```java
read("hdfs:rawMsgs")
-> transform {$.*, phones: findPhones($.msg)};
```

Express user code with:
- Java
- Inline scripting
- ...

Input data

```json
[
  { from: 17, to: 19, msg: "..." },
  { from: 17, to: 12, msg: "..." },
  { from: 17, to: 19, msg: "..." }
]
```

Output data

```json
[
  { from: 17, to: 19, msg: "...",
    phones: ["+1-415-555-1212",
             "+1-408-555-1234"] },
  { from: 17, to: 12, msg: "..."},
  { from: 17, to: 19, msg: "...",
    phones: ["+1-415-555-2345"] }
]
```
Inline Scripting

```python
import re
import phoneUtil

phoneRE = re.compile('^[+x]?[0-9\-()]\+\$')

def findPhones(text):
    for p in phoneRE.findall(text):
        p = phoneUtil.standardize(p)
        if p != None:
            yield p

END;

read("hdfs:rawMsgs")
-> transform {$.*, phones: findPhones($.msg)};
```

- Easily integrate with other programming languages
  - Python first
  - then JavaScript, Ruby, Perl, Unix shell, etc
Grouping

Find the number of times the each sender mentioned an area code

\[
\text{read(“hdfs:rawMsgs“)} \\
\text{-> expand for( $p \text{ in findPhones($msg) } ) \{ $.from, area: area($p) \} ]} \\
\text{-> group by $g = (\{$.from, $.area\})} \\
\text{into \{ $g.*, n: count($) \};}
\]

Input data

```
[ 
  { from:17, to: 19, msg:"..." },
  { from: 17, to: 12, msg:"..." },
  { from: 17, to: 19, msg:"..." }
]
```

After expand

```
[ 
  { from: 17, area: 415 },
  { from: 17, area: 408 },
  { from: 17, area: 415 }
]
```

Output data

```
[ 
  { from: 17, area: 415, n: 2 },
  { from: 17, area: 408, n: 1 }
]
```
Map-Reduce Jobs in Hadoop

\[
[ V ] \rightarrow \text{Map} \rightarrow [ K, V' ] \rightarrow \text{Shuffle} \rightarrow [ K, [V'] ] \rightarrow \text{Reduce} \rightarrow [ V'' ]
\]

\[
\begin{align*}
& M \\
& M \\
& M \\
\end{align*}
\]

$ \rightarrow \text{expand} \quad \text{for}( \ $p$ \in \text{findPhones}($msg$) \ ) \\
\quad [\{ \ $.from, \text{area: area}($p$) \ }] \\
\rightarrow \text{transform} [\{$.from, $.area\}, ]$

\quad \text{mapReduce}(\{
\begin{align*}
& \text{input} : \{\text{uri: “hdfs:rawMsgs”}\}, \\
& \text{map} : \text{fn}($) (\rightarrow \text{expand} \text{for}( \ $p$ \in \text{findPhones}($msg$) \ ) \\
\quad [\{ \ $.from, \text{area: area}($p$) \ }] \\
\quad \rightarrow \text{transform} [\{$.from, $.area\}, ]$, \\
& \text{reduce} : \text{fn}($g,$) ($ \rightarrow \text{transform} \{ \ $g.*, \text{n: count}($) \ }$, \\
& \text{output} : \{ \text{uri: “hdfs:out”}\} )
\})
Exploiting Combiners with Algebraic Aggregates

\[ [ V ] \rightarrow \text{Map} \rightarrow [ K, V'] \rightarrow \text{Shuffle} \rightarrow [ K, [V'] ] \rightarrow \text{Reduce} \rightarrow [ V'' ] \]

$ -> \text{expand}$

\[
\text{for}( \$p \text{ in } \text{findPhones}(\$.msg) ) \]

\[
\{ \$.from, \text{area: area}(\$p) \}\]

$ -> \text{transform}$

\[
\{\$.from, \$.area\}, \$
\]

$ -> <\text{initial aggregation}>$

\[
\text{mrAggregate}\{
\text{input} : \{\text{uri: “hdfs:rawMsgs”}\},
\text{map} : \text{fn}(\$) (\$ -> \text{expand} \text{for}( \$p \text{ in } \text{findPhones}(\$.msg) ) \]

\[
\{ \$.from, \text{area: area}(\$p) \}\]

\[
\text{transform} \{\$.from, \$.area\}, \$
\]

$ -> <\text{final aggregation}>$

\[
\text{transform} \{ \$g.*, \text{n: } \$[0]\}
\]

$ -> <\text{partial aggregation}>$
Composite Operators

Examples:

- **Join**
  - Join two or more inputs on a key
  - Inner/outer/full
  - Multi-predicate, multi-way

- **Merge**
  - Concatenate all inputs in any order

- **Union, Intersect, Difference**...

- **User-defined function**
CoGroup

Find users in each zipcode and the name of the city

```java
$users = read("hdfs:users");
$cities = read("hbase:cities") -> expand unroll $.zips;

group $users by $zip = ($users.zip),
    $cities by $zip = ($cities.zips)
into { $zip, n: count($users), users: $users[*].name, city: singleton($cities[*].city) };
```

Input data

**users**

```
[ { id: 12, name: "Joe Smith", zip: 94114 },
  { id: 17, name: "Ann Jones", zip: 95120 },
  { id: 19, name: "Alicia Fox", zip: 94114 } ]
```

**cities**

```
[ { city: "San Francisco", zips: [94112, 94114], areas: [415] },
  { city: "San Jose", zips: [95120], areas: [408] } ]
```

**after group**

```
[ { zip: 94114,
    n: 2,
    users: [ "Joe Smith", "Alicia Fox" ],
    city: "San Francisco" },
  { zip: 95120,
    n: 1
    users: [ "Ann Jones" ],
    city: "San Jose" },
  { zip: 94112,
    n: 0,
    users: [],
    city: "San Francisco" } ]
```
Multiple Outputs: tee

Send each input item to additional output pipe
read("hdfs:users")
→ **tee** (→ filter $.zip == 94114
→ write("hdfs:zip94114") )
→ filter $.bday > date("1979-01-01")
→ write("hdfs:over30");
### Rough Unix analogs of Jaql

<table>
<thead>
<tr>
<th>Unix</th>
<th>Jaql</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; filename, cat</td>
<td>read, merge</td>
</tr>
<tr>
<td>join</td>
<td>join</td>
</tr>
<tr>
<td>grep</td>
<td>filter</td>
</tr>
<tr>
<td>cut, paste, sed, tr</td>
<td>transform</td>
</tr>
<tr>
<td>sort</td>
<td>sort</td>
</tr>
<tr>
<td>head</td>
<td>top</td>
</tr>
<tr>
<td>uniq</td>
<td>distinct</td>
</tr>
<tr>
<td>&gt; filename</td>
<td>write</td>
</tr>
<tr>
<td>tee</td>
<td>tee</td>
</tr>
</tbody>
</table>

**Unix**: stream of bytes / lines

**Jaql**: stream of JSON items

more structure / types
Other Map/Reduce Languages

- **Java**
  - The most basic way to write a map/reduce job in Hadoop
  - User implements a Mapper and Reducer interface and creates a JobConf

- **Hadoop Streaming**
  - Map and Reduce tasks implemented by external process in any language
  - Read/write data on stdin/stdout

- **Pig**
  - Nested relational model, but added “Map” for dynamic columns
  - From Yahoo. Hadoop subproject.

- **Hive**
  - Nested relational model with “Map” for dynamic columns and “Array” for lists
  - SQL-like syntax
  - From Facebook. Hadoop subproject.

- **Cascading**
  - A dataflow language, mostly relational.
  - Expressions are UDFs; Groovy extension

- **Proprietary and non-Hadoop**
  - Google Sawzall, Microsoft DryadLINQ, Microsoft Scope
Current Roadmap

- New syntax (presented here)
  - About to be released to open source
- Robustness
  - Exception handling, …
- Features
  - Programming/scripting language integration, schema, tooling, …
- Performance
  - Query optimization, compact storage, runtime tuning, …